Compositional model checking of SDN platform

Abdul Majith, Ocan Sankur, Hervé Marchand, Thai Dinh

Compositional model checking of SDN platform

Abdul Majith
Univ Rennes, Inria, CNRS
Rennes, France
abdul-majith.noordheen@inria.fr

Ocan Sankur
Univ Rennes, Inria, CNRS
Rennes, France
ocan.sankur@irisa.fr

Hervé Marchand
Univ Rennes, Inria, CNRS
Rennes, France
herve.marchand@inria.fr

Dinh Thai Bui
Nokia Bell Labs France
Nozay, France
dinh_thai.bui@nokia-bell-labs.com

***Abstract*—Software-Defined Network (SDN) technology provides the possibility to turn the network infrastructure into a dynamic programmable fabric capable of meeting the application needs in real-time. Thanks to the independence of the control plane from the data plane, the control entity, generally called as controller, has also the flexibility to implement proprietary complex algorithms. Within such a dynamic and complex environment, this document advocates for applying formal verification methods and more precisely composition model checking to ensure the correct behavior of the overall SDN system at design phase. To illustrate this purpose, it proposes to build different comprehensive formal models of a typical SDN platform selected here as a study object. Thorough performance results related to each model are provided and discussed. Thanks to such formal verifications, it is possible to pinpoint issues such as the one regarding network isolation within a complex SDN architecture. Although dealing with formal methods, this document attempts to strike a balance between theory, experimental work and network architecture discussion.**

***Index Terms*—SDN, slice, compositional checking, verification, isolation, safety**

I. Introduction

The flexibilty of Software-Defined network (SDN) together with the implementation of new complex network algorithms raises new challenges in term of validation and verification. Formal verification methods which are capable of exhaustively checking the behaviors of a given system have been applied in the setting of SDNs (see Section II). We also advocate for applying formal verification methods to ensure the correct behavior of an SDN platform especially during the design phase. More precisely, we demonstrate that *compositional model checking* is well-adapted to model and verify SDN architectures with all their different layers or planes, namely the data plane, the control plane and the management plane. For this purpose, we propose to build complete models of a typical SDN architecture taken here as a study object. Compositional modeling allows for designing and verifying each layer separately from others leading to a more modular and more scalable verification mechanism. Our contribution is fourfold. First, we present a comprehensive formal model of the SDN platform published in [1]. We model respectively devices and their mobility, switches and the network topology on the data plane, controllers on the control plane and managers on the management plane. We model communications between components of the same planes (i.e. between devices and switches) as well as those in different planes. Second, we demonstrate that compositional reasoning is well-adapted to model-check SDN by decomposing the overall complex system verification into the sum of simpler verifications of the different components. We perform both a vertical (e.g. between planes) and a horizontal (e.g. between devices and switches) decomposition during model checking. Note that our approach consists in model checking the system model *offline*, that is, exhaustively checking all possible behaviors of the model before deployment. Third, we show that it is possible to verify complex scenarios such as to discover a network slice isolation violation. Finally, we propose a method and related architecture to avoid the issue previously discovered and verify that such a method is valid. After going through state-of-the-art in Section II, we describe the SDN platform of [1] taken as a study object in Section III. In Section IV we develop a complete formal model of the studied platform in the form of automata. An implementation of such automata is also proposed in Section V using a tool called Spin [2]. In the same section, we argue that compositional reasoning is well appropriate for SDN and propose the application of such reasoning to our previous formal model. Most of experimental works are carried out in Section VI where we discuss on performance of different verification methods according to model sizes while assuming devices are not moving (i.e. no mobility condition). The model with device mobility in Section VII allows us to pinpoint a network slice isolation issue. We finally propose a solution to correct the isolation violation and formally verify that the solution is valid.

II. Related Work

The closest to our work is that of [3] which presents a tool called Kuai. The latter applies optimizations based on partial-order reduction to simplify the models of SDN protocols, and presents several model checking benchmarks to assess the performance of its optimizations. The tool translates formal models given in the Murphi input format and uses PReach [4] for distributed model checking. One of its optimizations is based on *barriers* which are used to ensure the order in which switches implement SDN rules. We do not consider barriers in our platform since our SDN rules are applied asynchronously. Also, Kuai uses $(0, \infty)$-abstractions on packet queue contents which consist in storing whether a given packet is absent (0) or present an unknown number of times (∞). Since we present an abstract model by focusing on particular packets, we do not use this abstraction in our work. Our entire SDN platform is an asynchronous distributed system where various components

are with finite number of FIFO queues. There is no priority on queues while reading incoming messages. Instead, each component reads messages by performing queue polling in a round robin fashion. We can model the system mathematically as a synchronous one by encoding state space along with current queue message contents. In an asynchronous system, barriers optimizations and partial order techniques might be used but verifications have to be performed on such abstractions in order to check that they preserve the safety properties. Indeed, reading various messages among different incoming queues in different orders might lead to different results (see the case study described in Section VII).

With regards to experimental tools, we have selected a tool called as SPIN (Simple Promela INterpreter) which has the advantages of implementing in-build partial order techniques. It spares modelers from having to write specific codes to implement partial order techniques while specifying the behaviors of the SDN platform. As per our knowledge, the use of compositional reasoning for SDN protocols is new - e.g. compared to [3]. Moreover, we consider the SDN architecture with a management plane which renders the system to be analyzed more comprehensive and more complex. To our knowledge, this is the first time compositional reasoning is used to verifying SDN protocol correctness. However, this method has some limitations as one has to guess intermediate formula (See Section V) If such intermediate and global properties are only safety properties then finding such intermediate formulas can be automated for such automated procedure by using e.g. Learning ω-Regular languages [5].

Several other works consider the formal analysis of SDN protocols. [6] provides a static analysis of header spaces which detect inconsistencies in routing tables, but does not consider more dynamic behaviors and complex issues due to interleavings as it is done in [3] and in our work. A similar approach was adopted in [7]. A modeling language called FlowLog tailored for SDN protocols was given in [8] in which the authors present model checking experiments with the SPIN model checker [2] but no particular optimizations are presented. In [9], model checking and symbolic execution are combined in order to check OpenFlow applications for a given number of packets. [10] presents data and network abstractions applied on SDN models combined with a manual refinement process based on *non-interference lemmas*.

In [11], the authors develop a framework where the system is modeled using first-order logic, and user-provided inductive invariants are checked to prove correctness. This allows one to check the system for all network topologies, and for an unbounded number exchanged packets.

The approach developed in [12] consists in checking the effect of rule updates in real-time, without affecting the system performance. This is complementary to *offline* verification approaches which analyze the system globally before execution.

In [13], reactive synthesis is used to automatically synthesize network update rules in response to network requests. Other works consider the synthesis of update rules that makes sure that the intermediate configurations during the update are all admissible [14], [15], or compute correct network-wide configurations [16]. The procedure described in [14] allow to automatically synthesize network updates but are only described at a theoretical level and its deployment to real-time system procedure is difficult. Because, in the proposed solution, each subsystem need the approval of the global system in order to deploy new rules (ths is performed by the use of a so-called *wait* command. Such mechanism is possible in asynchronous system whenever the protocol have an authentication mechanism or at least timestamp needs to be added within the exchanged messages.

Abstractions and other transformations that preserve the properties of networks for rendering the verification more efficient are presented in [17]. In this work, we are interested to provide the methodology to use compositional reasoning to verify SDN platform correctness. Properties selected for verification are estimated by the authors as critical for a good behavior of an SDN platform. Naturally, compositional reasoning is a well-adapted verification analysis for SDN, since SDN platforms consist in a clear separation between control, management and data planes. In order to ensure that the SDN platform satisfies global properties, it is sufficient to verify that each plane satisfies specific local properties, which makes the model checking procedure scalable with respect to time and space complexity. This methodology also allows for analyzing and designing successively each individual component in a iterative manner of whole complex system.

III. The SDN Platform Description

In this document, we build a formal and abstract model of the SDN platform [1] which is taken here as a study object. We think that this SDN platform has a representative architecture to make our study as generic as possible.

A. Architecture Building Blocks

As most of SDN platforms, SDN platform [1] consists of three layers, namely a management plane, a control plane and a data plane. The management plane are formed of entities called 'managers'. Each manager has the role of interfacing with the user and of converting the user's intents into high level network policies. The control plane consists of entities called 'controllers' which have the role of converting managers' high level network policies into fine grain network rules and of enforcing those rules onto the data plane. The data plane consists of network elements embodied in the selected platform as Open vSwitches (OVS) [18]. OVSes are open-source OpenFlow switches [19] controlled by SDN controllers.

The platform is made of several administrative domains with each domain built up with one manager, one controller and several switches (i.e. OVSes). A given switch can only be part of one unique domain so that switches of all domains form a partition of the overall data plane. In order to simplify the controller algorithm in the forwarding of multicast/broadcast messages, the data plane topology of each domain is loop-free (the LLDP and the spanning-tree mechanisms should be added to the controller logics if non loop-free topologies are

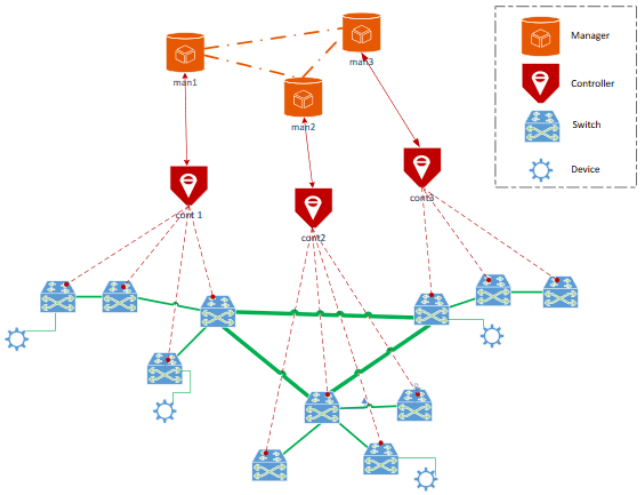

Fig. 1. SDN Platform Description

considered. This is part of our next steps). In this document, such a topology is a tree topology departing from a root switch. The latter is the domain border switch. Root switches of all domains are interconnected together in a full meshed topology so that there is only one network hop between any pairs of root switches. Figure 1 provides us with a schematic representation of the SDN platform.

The data plane is also completed with devices. Each of them can be connected to a switch port called as an *Access-port* via a network link called as an *Access-link* (thin green links). Similarly, switches within a domain are connected together via network links called as *Intra-links* (medium-size green links) thanks to their *Intra-ports*. Root switches from different domains are interconnected together via *Inter-links* (thick green links) thanks to their *Inter-ports*.

B. The User's Intent

The user's intent consists in a set of predicates on device characteristics or device capabilities [20]. This set of predicates or intent allows the user to select devices and gather them into a private group called in the following sections as *Virtual Space* or *VS*. Devices of the same *VS* are to be connected together by controllers on the different domains where these devices are detected (i.e. via the MAC learning process). A *VS* is enforced by the control plane as a network slice in the data plane. While enforcing OpenFlow rules, controllers should make sure that a pair of devices which do not belong to any common *VS* are not connected together. This is defined as the network slice isolation property or isolation property for short (i.e. data traffic isolation between network slices) in this document. For the sake of simplicity, the user's intent will be presented in the rest of the document as a nominative list of devices - e.g. $\{d1, d2, d3\}$. This is done without losing the genericity of our discussions since the present verification work is not focusing on the consistency of the overall users' intents but rather on the safety of the underlying SDN mechanisms. Moreover, we do not present details on how the controller which identifies a given device by its MAC address, and the associated manager which identifies the same device by its name, agree with each other on the device identity, as per the limited size of the document. Please refer to [1] for further information.

C. Device Discovery via MAC Learning

The controller implements the well-known MAC learning mechanism as per [21]. Thanks to this mechanism, the controller can discover newly connected device (e.g. new MAC address) and inform the associated manager. In return, it receives from the manager high level network policies (derived from users' intents) regarding the new device.

Once the new device has been discovered by the domain controller, the latter 'leaks' multicast/broadcast packets from the former to all neighbor domains where devices of the same VS are located (and only to those domains). Such packets are treated as MAC learning packets in such neighboring domains, meaning that they are flooded within the aforementionned domains except to their *Access-ports*. The leaking process is triggered thanks to knowledges coming from managers which synchronize between them, VS predicate and device location. In order to avoid forwarding loops during the leaking process, controllers apply a simple algorithm known as the *split horizon* algorithm. Within such an algorithm, a root switch never forwards to other root switches a packet it has received from another root switch. The *split horizon* algorithm does properly ensure reachability between domain given that we have a full mesh topology between root switches.

The work in this document is modeling the above device discovery procedure through MAC learning at all planes.

D. Packet Forwarding

A unicast packet with source MAC address 'S' and destination MAC address 'D' is forwarded if MAC addresses 'S' and 'D' are considered as part of the same *VS* (assuming the right mapping between MAC addresses and device names as discussed previously). Otherwise, the packet is dropped.

Concerning multicast (including broadcast) packets, the controller derives a multicast tree for each source MAC address 'S' [22] based on the different *VSes* this MAC address 'S' belongs to. In order to avoid network loops when forwarding multicast packets over Inter-links, each domain controller implements the well-known *split horizon* algorithm taking into account the full mesh nature of the *Inter-links* topology. The *split horizon* algorithm simply consists in not forwarding to other *Inter-links* a packet coming from an *Inter-link*.

Apart from multicast packet forwarding, all the data plane including unicast forwarding and flooding mechanism which is part of the MAC learning process is modelled during this work. As it concerns the control plane and the management plane, their essential behaviors are completely captured by the models described in this document.

IV. Automata-based Formal Models

We formalize our model using finite automata. We start by introducing the formalism, and then describe the formal model of SDN platform.

A. Preliminaries

Definition IV.1. Automaton: *an automaton is a tuple of the form $(Q, q_{init}, \Sigma \cup \{\tau\}, \Delta, L)$, where Q is the finite set of states, $q_{init} \in Q$ is the initial state, $\Sigma = \Sigma_{in} \cup \Sigma_{out}$ is the alphabet and τ a distinguished symbol for internal transition, $\Delta : Q \times \Sigma \cup \{\tau\} \to Q$ is the transition relation, and $L : Q \to 2^{AP}$ is a labeling function with AP a given set of atomic predicates.*

A transition $\delta = (q, \sigma, q') \in \Delta$ will also be written as $q \xrightarrow{\sigma} q'$ if δ is clear from the context. Moreover, for better readability, we will use $\sigma!$ if $\sigma \in \Sigma_{\text{out}}$, and $\sigma?$ if $\sigma \in \Sigma_{\text{in}}$. This is only to help the reader; formally the label does not contain the symbols ? and !.

Definition IV.2. Synchronized product of automata: *given automata $\mathcal{A}_i = (Q^i, q^i_{init}, \Sigma^i \cup \{\tau\}, \Delta^i, L^i)$ for $i \in \{1, \ldots, n\}$, the* synchronized product *of $\mathcal{A}_i$ is defined as $\mathcal{A} = \mathcal{A}_1 \parallel \ldots \parallel \mathcal{A}_n = (\Pi^n_{i=1} Q^i, (q^1_{init}, \ldots, q^n_{init}), \Sigma \cup \{\tau\}, \Delta, L)$ where $\Sigma = \cup^n_{i=1} \Sigma_i$, $L : \Pi^n_{i=1} Q^i \to (2^{AP_1}, .., 2^{AP_n})$ s.t $L(q_1, q_2, .., q_n) = (L^1(q_1), .., L^n(q_n))$, and Δ is defined as follows. For all states $(q_1, \ldots, q_n) \in \Pi^n_{i=1} Q^i$,*

- *for all $1 \leq i \leq n$, and $q'_i \in Q^i$ such that $\Delta^i(q_i, \tau, q'_i)$, $((q_1, \ldots, q_i, \ldots, q_n), \tau, (q_1, \ldots, q'_i, \ldots, q_n)) \in \Delta$,*
- *for all $\sigma \in \Sigma$, writing $I_\sigma = \{1 \leq i \leq n \mid \exists q'_i \in Q^i, \Delta^i(q_i, \sigma, q'_i)\}$, $(q_1, \ldots, q_n) \xrightarrow{\sigma} (q'_1, \ldots, q'_n)$ if, and only if for all $i \in I_\sigma$, $\Delta^i(q_i, \sigma, q'_i)$, and for all $i \notin I_\sigma$, there exists no q'_i such that $\Delta^i(q_i, \sigma, q'_i)$.*

In our automata, the set AP contains predicates $\text{Sync}(\sigma)$ for all labels $\sigma \in \Sigma$, such that $\text{Sync}(\sigma)$ is true iff the previous transition was labeled by σ. Note that these predicates appear in several components but all states of the product agree on their values by definition of the synchronized product.

B. Important Concepts

An instance of the platform is defined by providing the topology for different planes. Recall that there is exactly one manager associated to each controller. Each controller only communicates with its own manager and the switches in its domain. Managers are interconnected together via a complete graph of communication.

Formally, consider a set of controllers $\text{Cont} = \{c_1, \ldots, c_k\}$, a set of managers $\text{Man} = \{\text{man}_1, \ldots, \text{man}_k\}$ with the same cardinal, and a set of switches $\text{Sw} = \text{Sw}_1 \cup \ldots \cup \text{Sw}_k$ given as a partition. Within the selected platform, the switches of Sw_i belong to the *domain* i which is controlled by the controller c_i. The latter is itself managed by man_i. We consider a set Dev of devices which can connect to switches of different domains.

In each domain i, there is one designated switch called the *root switch*, denoted by $\text{root}_i \in \text{Sw}_i$. The data plane topology is a graph $G = (\text{Sw}, E)$ such that the subgraph restricted to each Sw_i is an undirected tree rooted at root_i, and the subgraph induced by the set of roots $\{\text{root}_1, \ldots, \text{root}_k\}$ is a complete graph.

In our studied platform, we distinguish so-called *Access-ports* through which devices connect to switches. We also have *Intra-ports* which are used to interconnect switches that belong to the same domain, and *Inter-ports* which are used to interconnect *root switches* of different domains. We also define a finite set Pts which refer to the set of *Access-ports* available at all switches.

Platform instance: an instance of the studied platform is defined as a tuple $(\text{Man}, \text{Cont}, \text{Sw}, G)$ where:

- $\text{Man} = \{\text{man}_1, \ldots, \text{man}_k\}$,
- $\text{Cont} = \{c_1, \ldots, c_k\}$,
- $\text{Sw} = \bigcup_{i=1}^{k} \text{Sw}_i$,
- $G = (\text{Sw}, E)$ is the topology graph as explained above.

We also use the function $\text{cont} : \text{Sw} \to \text{Cont}$ which identifies the controller associated to each switch, thus determining its domain: for all $i \in \{1, \ldots, k\}$ and $v \in \text{Sw}_i$, we have $\text{cont}(v) = c_i$.

Virtual Spaces (VSes): we consider the covering of Dev by the set of *Virtual Spaces* $\mathsf{VS} = \{V_1, V_2, .., V_f\}$, with $V_i \subseteq \text{Dev}$ for all $1 \leq i \leq f$, and $\cup^l_{i=1} V_i = \text{Dev}$. Let us define the function $\mathsf{VS}(x) = \{V \in \mathsf{VS} \mid x \in V\}$ which assigns to each device x the set of VSes to which x belongs.

Exchanged Data Packets: two types of packets are exchanged between switches and devices. *MAC learning packets* are sent by devices and forwarded by switches, and they contain the device identifier; while *ping packets* contain the source and target device identifiers.

These are defined, respectively, as $\text{MacPkts} = \{\mathsf{mac}\} \times \text{Dev}$ and $\text{PingPkts} = \{\mathsf{ping}\} \times \text{Dev} \times \text{Dev}$. These packets will be written as $\mathsf{mac}(x)$ and $\mathsf{ping}(x, y)$ respectively, for $x, y \in \text{Dev}$.

Exchanged Control Packets: let us define $\text{ManPkts} = \text{Dev} \times \{1, .., k\} \times 2^{\mathsf{VS}}$ a set of packets from the management plane to the controller plane and $\text{ContPkts} = \{(\text{man}_i, c_i) | i \in \{1, .., k\}\} \times \text{Dev}$ is the set of packets from the controller plane to the management plane.

It is noted that MacPkts forwarded by switches to the associated controllers are also part of exchanged control packets.

OpenFlow Rules: controller sends two types of rule updates to switches.

Rules impacting the forwarding of ping packets are defined as $\text{PingRules} = \text{Dev} \times \text{Dev} \times (\text{Pts} \cup \text{Sw})$ where the triple contains the source and target devices identifiers and the port or the switch to forward to.

Rules impacting the forwarding of MAC learning packets are $\text{MacRules} = \text{Dev} \times \text{Sw}$ where the pair contains the identifier of the device that has generated the mac learning packet, and the switch to forward to.

We let $\text{Rules} = \text{PingRules} \cup \text{MacRules}$.

Notice that we use switch identifiers as destinations instead of *Intra-port* identifier or *Inter-port* identifier to which both ping and MAC packets are forwarded to. This is because switch identifiers are used as port numbers for the communication between switches (see more detailed in section IV-C2).

C. Automata Models

We now describe the different automata that respectively model devices, switches, controllers, and managers. We completely define the related state space but only provide descriptive information on the related transitions.

We denote the set of words on the alphabet A by $\mathrm{Seq}(A) = A^*$, while the empty word is ϵ. This notation will be used to encode packet queues in the automaton below. Note that we describe unbounded packet queues in our formal model, although the real platform uses bounded ones (see below).

1) Automaton for Devices: We define an automaton $\mathcal{A}^{\mathrm{Dev}}$ that describes the behaviors of the whole set of devices Dev. The automaton stores the positions (i.e. the *Access-port* which a device is connected to) of all devices, transmits MAC learning and ping packets, receives pings packets and allows devices to change positions. Moreover, the automaton also stores the set of ping packets that have been sent, and the set of ping packets that have been received by each device since the start. This information is used to check whether transmitted and received packets match the verification specifications. We restrict to the case where each device pings at most once any other device.

The device state space contains:

- a position function **pos** of type $\mathrm{Dev} \to (\mathrm{Pts} \times \mathrm{Sw}) \cup \{\bot\}$ which assigns each device a switch and an *Access-port* if it is connected, and $\bot$ otherwise;
- a set **spings** $\subseteq$ PingPkts which stores sent ping packets;
- a set $\mathbf{rpings}_x \subseteq$ PingPkts each device x which stores received ping packets.

Initially, these spings and rpings_x for all $x \in$ Dev sets are empty, and pos maps all devices to $\bot$. Intuitively, a device x can send ping or MAC learning packets m to the switch v through port p (this has the form $(x, v, p, (\mathsf{mac}, x))$ or $(x, v, p, (\mathsf{ping}, x, y))$), and they can receive ping packets m from switch v via port p (this has the form $(v, p, (\mathsf{ping}, z, x))$). The latter does not contain the device identifier. In fact, this packet is forwarded by OVS v to port p, and any device that is connected to this port at that moment receives the packet.

Moreover, a device sends a new MAC learning packet whenever it changes its position.

The automaton $\mathcal{A}^{\mathrm{Dev}}$ has the following transitions.

► Changing position and sending a MAC learning packet:
$q \xrightarrow{(x,v,p,\mathsf{mac}(x))!} q\,[\mathrm{pos} \leftarrow \mathrm{pos}[x \mapsto (v,p)]]$,
for all $p \in \mathrm{Pts}, v \in \mathrm{Sw}, x \in \mathrm{Dev}$.

► Sending a ping packet:
$q \xrightarrow{(x,v,p,\mathsf{ping}(x,y))!} q\,[\mathrm{spings} \leftarrow \mathrm{spings} \cup \{\mathsf{ping}(x,y)\}]$,
for all $x \in \mathrm{Dev}$ and $\mathsf{ping}(x,y) \notin \mathrm{spings}$,

► Receiving a ping packet:
$q \xrightarrow{(v,p,\mathsf{ping}(x,y))?} q\,[\mathrm{rpings}_z \leftarrow \mathrm{rpings}_z \cup \{\mathsf{ping}(x,y)\}]$
for all $v \in \mathrm{Sw}, p \in \mathrm{Pts}$ and $\mathrm{pos}(z) = (v,p)$.

2) Automaton for Switches: Here we define an automaton $\mathcal{A}^{\mathrm{Sw}}$ that captures the behaviors of all the switches. Remember that we use switch identifiers as destinations instead of *Intra-port* identifier or *Inter-port* identifier to which both ping and MAC packets are forwarded to. This is because switch identifiers are used as port numbers for the communication between switches.

For simplicity of modeling and to reduce the state space, we abstract away from port numbers for communication between switches, and use switch identifiers instead. For instance, while flow tables are used to map packets to ports to forward to in the real system, our flow tables map packets to switch identifiers except when the outgoing port is an *Access-port*. This is without loss of generality since a switch identifier determines the port to which it is connected and vice versa (once the topology between switches is defined). Moreover, as mentioned in the beginning of this section, we omit outgoing packet queues, and assume that sending a packet consists in writing directly in the incoming packet queue of the recipient switch.

The automaton contains for each switch $v \in$ Sw the following components:

- ping packet forwarding rules $\mathbf{pfwd}_v : \mathrm{Dev} \times \mathrm{Dev} \to \mathrm{Sw} \cup \mathrm{Pts} \cup \{\bot\}$ that is a partial function which, for a given ping packet $\mathsf{ping}(x,y)$ from device x to device y, assigns a switch or an *Access-port* to forward to;
- MAC learning packet forwarding rules $\mathbf{mfwd}_v : \mathrm{Dev} \to 2^{\mathrm{Sw}}$ which gives the set of switches to which to forward the MAC learning packet received from given device;
- a data packet queue $\mathbf{dque}_v$ of type $\mathrm{Seq}(\mathrm{MacPkts} \cup \mathrm{PingPkts})$ to store packets received from other switches;
- and a control packet queue $\mathbf{cque}_v$ of type $\mathrm{Seq}(\mathrm{Rules})$ that stores rule packets coming from the controller.

The switches receive MAC learning or ping packets from devices (packets of the form (x, v, p, m)) which are put to dque_v. They forward MAC learning packets they have received to the controller (packets of the form $((v, \mathrm{cont}(v)), (p, \mathsf{mac}(x))))$, and receive new rule updates from the controller (packets of the form $(\mathrm{cont}(v), v, r)$), which are put into cque_v.

Initially, pfwd_v is empty, and mfwd_v maps all devices to the set of neighboring switches of the same domain, that is, $\mathrm{mfwd}_v(x) = \{v' \in \mathrm{Sw}_i \mid \mathrm{cont}(v) = \mathrm{cont}(v') \wedge (v,v') \in E\}$ for all $x \in$ Dev. Moreover, all queues dque_v and cque_v are empty at the initial state.

When processing a MAC rule, the process pops a packet (x,p) from the queue cque_v, updates and triggers its mfwd_v function to forward MAC learning packets for device x to all neighboring v' switches. Similarly, when processing a ping rule (x,y,p), the flow table pfwd_v function is updated and triggered so that ping packets from x to y are forwarded to p.

The process forwards MAC learning packets $(p, \mathsf{mac}(x))$ to all neighbor switches v' (except the switch p where the packet comes from) in $\mathrm{mfwd}_v(x)$, but it also forwards it to the controller, via the synchronizing transition (with label $((v, \mathrm{cont}(v)), (p, \mathsf{mac}(x)))$.

When a *root switch* receives a MAC learning packet from another *root switch* (i.e. of another domain), the former only forwards the packet to its domain as per the previously described *split horizon* principle in order to avoid loops.

Forwarding ping packets are internal transitions. A packet $\mathsf{ping}(x,y)$ is forwarded by $\mathrm{pfwd}_v(x,y)$ if this value is defined otherwise the packet is dropped. Transitions for $\mathcal{A}^{\mathrm{Sw}}$ are defined as follows.

▶ Receiving a packet from the controller:
$q \xrightarrow{((\mathrm{cont}(v),v),r)?} q\,[\mathrm{cque}_v \leftarrow \mathrm{cque}_v \cdot r]$, for all $r \in \mathrm{Rules}$

▶ Receiving a MAC learning packet from a device:
$q \xrightarrow{(x,v,p,\mathsf{mac}(x))?} q\,[\mathrm{dque}_v \leftarrow \mathrm{dque}_v \cdot (p,\mathsf{mac}(x))])$,
for all $x \in \mathrm{Dev}, p \in \mathrm{Pts}$.

▶ Receiving a ping packet from a device:
$q \xrightarrow{(x,v,p,m)?} q\,[\mathrm{dque}_v \leftarrow \mathrm{dque}_v \cdot m])$,
for all $x \in \mathrm{Dev}, p \in \mathrm{Pts}, m \in \mathrm{PingPkts}$,

▶ Processing a MAC rule: $q\,[\mathrm{cque}_v = (x,v') \cdot \mathrm{cque}_v] \xrightarrow{\tau}$
$q\,[\mathrm{mfwd}_v \leftarrow \mathrm{mfwd}_v[x \mapsto \mathrm{mfwd}_v(x) \cup \{v'\}]]$,

▶ Processing a ping rule: $q\,[\mathrm{cque}_v = (x,y,p) \cdot \mathrm{cque}'_v] \xrightarrow{\tau}$
$q\,[\mathrm{pfwd}_v \leftarrow \mathrm{pfwd}_v[(x,y) \mapsto p], \mathrm{cque}_v \leftarrow \mathrm{cque}'_v]$,
for all $x,y \in \mathrm{Dev}, p \in \mathrm{Pts} \cup \mathrm{Sw}$.

▶ Forwarding a MAC learning packet:
$q\left[\mathrm{dque}_v = (p,\mathsf{mac}(x)) \cdot \mathrm{dque}'_v\right] \xrightarrow{((v,\mathrm{cont}(v)),(p,\mathsf{mac}(x)))!}$
$q\big[\mathrm{dque}_v \leftarrow \mathrm{dque}'_v, \forall v' \in S',$
$\mathrm{dque}_{v'} \leftarrow \mathrm{dque}_{v'} \cdot (v,\mathsf{mac}(x))\big]$,
where $p \in \mathrm{Pts} \cup \mathrm{Sw}$; if p is not a root switch $root_j$, then $S' = \mathrm{mfwd}_v(x) \setminus \{p\}$, else $S' = \mathrm{mfwd}_v(x) \cap \mathrm{Sw}_i$ with $i \neq j$.

▶ Forwarding a ping (to OVSs):
$q\left[\mathrm{dque}_v = \mathsf{ping}(x,y) \cdot \mathrm{dque}'_v\right] \xrightarrow{\tau}$
$q\left[\mathrm{dque}_{v'} \leftarrow \mathrm{dque}_{v'} \cdot \mathsf{ping}(x,y), \mathrm{dque}_v \leftarrow \mathrm{dque}'_v\right]$,
for all $v' = \mathrm{pfwd}_v(x,y) \in \mathrm{Sw}$,

▶ Forwarding a ping (to devices):
$q\left[\mathrm{dque}_v = \mathsf{ping}(x,y) \cdot \mathrm{dque}'_v\right] \xrightarrow{(y,v,p,\mathsf{ping}(x,y))!}$
$q\left[\mathrm{dque}_v \leftarrow \mathrm{dque}'_v\right]$, for all $p = \mathrm{pfwd}_v(x,y) \in \mathrm{Pts}$,

Note that in our model, packet communication between switches is modeled by directly writing in the packet queues of receiving switches.

The state space of the controllers also have Boolean variables toggle to indicate whether a ping update message has been sent to switches already.

3) Automaton for Controllers: We define automaton $\mathcal{A}^{\mathrm{Cont}}$ that describes the behaviors of the entire control plane.

The state space contains for each controller c_i the following components:

- a *device relative position* function $\mathbf{rpos}_i$ of type $\mathrm{Dev} \times \mathrm{Sw}_j \rightharpoonup \mathrm{Pts} \cup \mathrm{Sw} \cup \{\bot\}$ which gives the position of a given device at the given switch as known to the controller c_i, that is, the port from which the MAC learning packet from the device is received and to which ping packets for the device are to be forwarded at that switch.
- a *device information function* $\mathbf{dinfo}_i$ of type $\mathrm{Dev} \rightharpoonup \{1,\ldots,k\} \times 2^{\mathsf{VS}}$ which stores the domain and Virtual Spaces to which each device belongs;
- a queue $\mathbf{cdque}_i$ of type $\mathrm{Seq}(\mathrm{Sw}_i \times \mathrm{Pts} \times \mathrm{MacPkts})$ that stores packets that arrive from the data plane,
- and a queue $\mathbf{mque}_i$ of type $\mathrm{Seq}(\mathrm{ManPkts})$ that stores packets that arrive from the management plane.

Initially, partial functions rpos_i and dinfo_i, and both queues are empty.

Packets that come from the management plane are put in mque_i, while MAC learning packets from the data plane are put in cdque_i. When processing a MAC learning packet $(v,p,\mathsf{mac}(x))$, the controller updates its rpos_i function to store the relative position p of the device x at a given switch v. When processing a packet (x,dom,V) from the management plane, the controller updates its dinfo_i function to store the fact that device x belongs to domain dom and VS V. Furthermore, the controller sends ping rule updates of the form (x,y,p) to the switches whenever devices x and y belong to a common VS, and $\mathrm{rpos}_i(y,v) = p$. This ensures that ping packets with destination y be forwarded to p at switch v. Finally, the controller also sends MAC learning rule updates to *root switches*. It sends the rule (x,root_j) to switch root_i whenever device x is known to belong to domain i and there exists some device y in domain j which share a common VS with x. This ensures that MAC learning packets will be forwarded from domain i to domain j.

The transitions of $\mathcal{A}^{\mathrm{Cont}}$ are defined as follows.

▶ Receiving a packet from the management plane:
$q \xrightarrow{(\mathrm{man}_i,c_i,(x,dom,V))?} q\,[\mathrm{mque}_i \leftarrow \mathrm{mque}_i \cdot (x,dom,V)]$
for all $x \in \mathrm{Dev}, 1 \leq dom \leq k, V \subseteq \mathsf{VS}$.

▶ Receiving a MAC packet from the data plane:
$q \xrightarrow{((v,c_i),(p,\mathsf{mac}(x)))?} q\,[\mathrm{cdque}_i \leftarrow \mathrm{cdque}_i \cdot (v,p,\mathsf{mac}(x))]$,
for all $x \in \mathrm{Dev}, v \in \mathrm{Sw}_i$, $p \in \mathrm{Pts} \cup \mathrm{Sw}$.

▶ Processing a Mac learning packet (from IOT port):
$q\left[\mathrm{cdque}_i = (v,p,\mathsf{mac}(x)) \cdot \mathrm{cdque}'_i\right] \xrightarrow{(c_i,\mathrm{man}_i,x)!}$
$q\left[\mathrm{rpos}_i \leftarrow \mathrm{rpos}_i[(x,v) \mapsto p], \mathrm{cdque}_i \leftarrow \mathrm{cdque}'_i\right]$,
toggle $=$ false, for all $v \in \mathrm{Sw}_i, p \in \mathrm{Pts}$.

▶ Processing a Mac learning packet (from Non-IOT port):
$q\left[\mathrm{cdque}_i = (v,p,\mathsf{mac}(x)) \cdot \mathrm{cdque}'_i\right] \xrightarrow{\tau}$
$q\left[\mathrm{rpos}_i \leftarrow \mathrm{rpos}_i[(x,v) \mapsto p], \mathrm{cdque}_i \leftarrow \mathrm{cdque}'_i\right]$,
toggle $=$ false, for all $v \in \mathrm{Sw}_i, p \in \mathrm{Sw}$.

▶ Processing a packet from man_i:
$q\,[\mathrm{mque}_i = (x,\mathsf{dom},V) \cdot \mathrm{mque}'_i] \xrightarrow{\tau}$
$q\,[\mathrm{dinfo}_i \leftarrow \mathrm{dinfo}_i[x \mapsto (\mathsf{dom},V)], \mathrm{mque}_i \leftarrow \mathrm{mque}'_i]$,
toggle $=$ false, for all $x \in \mathrm{Dev}$, $1 \leq \mathsf{dom} \leq k$ and $V \subseteq \mathsf{VS}$.

▶ Sending ping rule updates to switches:
$q\,[\mathrm{toggle} == \mathrm{false}] \xrightarrow{(c_i,v,(x,y,p))!} q\,[\mathrm{toggle} = \mathrm{true}]$
for all $x,y \in \mathrm{Dev}, v \in \mathrm{Sw}_i$, $\mathrm{rpos}_i(y,v) = p \neq \bot$, and writing $\mathrm{dinfo}_i(x) = (j_x,V)$, $\mathrm{dinfo}_i(y) = (j_y,V')$ and $V \cap V' \neq \emptyset$.

▶ Sending MAC rule update to root_i:
$q \xrightarrow{(c_i,\mathrm{root}_i,(x,\mathrm{root}_j))!} q$ for all $x \in \mathrm{Dev}$ such that $\mathrm{rpos}_i(x,\mathrm{root}_i) \neq \bot \wedge \mathrm{dinfo}_i(x) = (i,V)$, and there exists $y \in \mathrm{Dev}$ s.t. $\mathrm{dinfo}_i(y) = (j,V')$ with $j \neq i$ and $V \cap V' \neq \emptyset$.

4) Automaton for Managers: We define automaton $\mathcal{A}^{\mathrm{man}}$ that describes the behaviors of the entire management plane. The state space contains for each management node man_i the following components:

- *a mapping function* $\mathbf{dinfo}_i : \text{Dev} \to \{1, \ldots, k\} \cup \{\perp\}$ determining the domain in which the device is connected.
- a packet queue $\mathbf{cque}_i$ that stores packets that come from c_i;
- and a queue $\mathbf{mque}_i$ for packets from other management nodes.

Initially, both queues are empty and dinfo_i maps all devices to $\perp$.

The management node man_i learns about the domains to which devices belong through packets received from the controller c_i, and forwards this information to all other management nodes. More precisely, packets sent by the controllers are put in the queues cque_i. When processing a controller packet for device $x \in \text{Dev}$, the manager man_i sends the packet (x, i) to all other managers informing them that device x belongs to domain i. This helps all management nodes to eventually have full information on the domain where each device is located. Moreover, the management node man_i sends to the controller c_i the VSes to which each known device belongs by sending packets of the form (x, dom, V) where x is a device, dom its domain, and V the set of VSes to which the device x belongs. The transitions of $\mathcal{A}^{\text{man}}$ are defined as follows.

► Receiving controller packet:
$q \xrightarrow{(c_i, \text{man}_i, x)?} q\,[\text{cque}_i \leftarrow \text{cque}_i \cdot x]$. where $x \in \text{Dev}$

► Processing a controller packet:
$q\,[\text{cque}_i = x \cdot \text{cque}'_i] \xrightarrow{\tau}$
$q\big[\forall j \neq i, \text{mque}_j \leftarrow \text{mque}_j \cdot (i, x),$
$\text{dinfo}_i \leftarrow \text{dinfo}_i[x \mapsto i], \text{cque}_i \leftarrow \text{cque}'_i\big]$.
where $x \in \text{Dev}$ and $1 \leq j \leq k$.

► Processing a management packet:
$q\,[\text{mque}_i = (j, x) \cdot \text{mque}'_i] \xrightarrow{\tau}$
$q\,[\text{dinfo}_i \leftarrow \text{dinfo}_i[x \mapsto j], \text{mque}_i \leftarrow \text{mque}'_i]$.

► Sending device information to controller:
$q \xrightarrow{(\text{man}_i, c_i, (x, j, V))!} q$ for all $x \in \text{Dev}, 1 \leq j \leq k$ such that $\text{dinfo}(x) = j$ and $V = \mathsf{VS}(x)$.

V. Formal Verification

During the following work, we use linear temporal logic (LTL) [23] to specify different system properties including the ones that we want to verify. LTL is build over propositional logic (based on Boolean operators $\neg, \vee, \wedge$) by adding the notion of discrete (*i.e* natural number) timeline and temporal operators such as the following $\{\,\Box,\, \Diamond\,\}$ just to name the ones we use here:

- $\Box\psi$ means ψ is True in **all** future moments
- $\Diamond\psi$ means ψ is True in **some** future moments

LTL formulas are built over a finite set of propositional variables referred to as *Atomic Propositions* (AP) combined with aforementioned Boolean and temporal operators. Temporal operators can be nested. For instance $\Box\Diamond\phi$ refers to executions in which at all moments, there is a moment in future where ϕ holds; thus, ϕ holds at infinitely many points in time. Formula $\Diamond\Box\phi$ refers to executions in which there is a moment in time after which ϕ holds at all positions; thus ϕ becomes an invariant after a while. We use the *finite trace* semantics of LTL [24] without next operators denoted by $\mathsf{LTL}_f \setminus \mathsf{X}$.

A. Property Specifications

We verify two important properties of our platform. The first is a *safety* property, called as the *isolation* property. It states that only devices that belong to a common VS can exchange ping packets and a device not belonging to a particular VS cannot eavesdrop ping packets exchanged among that VS group. This can be expressed in LTL as follows.

$$\mathsf{Isolation} = \Box\Big(\bigwedge_{\substack{x,y \in \text{Dev}, \\ \mathsf{VS}(x) \cap \mathsf{VS}(y) = \emptyset}} \mathsf{ping}(x,y) \notin \text{rpings}_y \;\; \bigwedge_{\substack{x,y,z \in \text{Dev}, \\ z \neq y}} \mathsf{ping}(x,y) \notin \text{rpings}_z \Big).$$

This LTL formula thus describes the set of executions which never enter a state where a packet $\mathsf{ping}(x, y)$ with $\mathsf{VS}(x) \cap \mathsf{VS}(y) = \emptyset$ is received, and no device can eavesdrop the packet which as to be received by different devices.
The second property is called as the *connectivity* property. It states that whenever a ping packet is sent to a device that belongs to a same VS group, the packet is eventually received; however this property only holds after some point in time, that is, when the MAC learning algorithm has finished. So the SDN platform eventually allows a common VS group to exchange the data packets among themselves. This can be expressed in LTL as follows:

$$\mathsf{Connect} = \bigwedge_{\substack{x,y \in \text{Dev} \\ \mathsf{VS}(x) \cap \mathsf{VS}(y) \neq \emptyset}} \Diamond\big(\mathsf{ping}(x,y) \in \text{spings} \wedge \Diamond \mathsf{ping}(x,y) \in \text{rpings}_y\big).$$

In our work we consider two scenarios. In the *no-mobility* scenario, devices do not change their respective position once they are connected to an *Access-port*. In the *mobility* scenario, devices can change their position at any time.
The first scenario called as Nomobility can be expressed in LTL as follows:

$$\mathsf{Nomobility} = \bigwedge_{\substack{x \in \text{Dev}, v \in \text{Sw}, \\ p \in \text{Sw} \cup \text{Pts}}} \Box\big((v,p) = \text{pos}(x) \rightarrow \Box(v,p) = \text{pos}(x)\big)$$

B. Compositional Reasoning

We present here a compositional reasoning methodology which allows one to model check a system by decomposing the problem into simpler verification tasks on each of its components. Such compositional techniques appeared early in the model checking community [25]–[27]. We also refer the reader to [28] where existing rules and algorithms are summarized.

Given an automaton $\mathcal{A}$ and LTL formulas ϕ, ψ, let the triple $\langle\phi\rangle\mathcal{A}\langle\psi\rangle$ denote $\mathcal{A} \models \phi \rightarrow \psi$. Intuitively, ϕ represents the *assumption* we make on the environment of $\mathcal{A}$, while ψ is the *guarantee* that $\mathcal{A}$ gives provided that the environment satisfies ϕ. We state the main theorem we use as follows.

Theorem V.1. *Consider automata $\mathcal{A}_1, \mathcal{A}_2$ with atomic propositions AP_1, AP_2 and alphabets Σ_1, Σ_2, and $\mathsf{LTL}_f \setminus \mathsf{X}$ formulas*

ϕ_1, ϕ_2, ϕ_3 *with* $\Sigma_{\phi_1} \subseteq \Sigma_1$, $\Sigma_{\phi_2} = \Sigma_1 \cap \Sigma_2$, *and* $\Sigma_{\phi_3} \subseteq \Sigma_2$. *We have the following rule.*

$$\frac{\langle\phi_1\rangle\mathcal{A}_1\langle\phi_2\rangle \qquad \langle\phi_2\rangle\mathcal{A}_2\langle\phi_3\rangle}{\langle\phi_1\rangle\mathcal{A}_1 \parallel \mathcal{A}_2\langle\phi_3\rangle}.$$

Thus, in order to prove $\langle\phi_1\rangle\mathcal{A}_1 \parallel \mathcal{A}_2\langle\phi_3\rangle$ it suffices to find a formula ϕ_2 and prove the two assertions $\langle\phi_1\rangle\mathcal{A}_1\langle\phi_2\rangle$ and $\langle\phi_2\rangle\mathcal{A}_2\langle\phi_3\rangle$ as per the previous theorem.

A proof of this theorem for *safety* properties can be found in [28]. Our formulas are general $\mathsf{LTL}_f \setminus \mathsf{X}$ formulas in the finite trace semantics. However, in our model of the SDN protocol, all executions eventually stop so that we can encode all our formulas as safety conditions using the Promela keyword `timeout` (which detects the end of an execution). For completeness, we do provide a proof of this theorem for general $\mathsf{LTL}_f \setminus \mathsf{X}$ formulas which can be applied to general models. This proof as follows.

The language of $\mathcal{A}$ denoted as $\mathcal{L}(\mathcal{A}) := \{\, \sigma \mid \sigma \in (\Sigma)^* \wedge \emptyset \neq q_0\Delta^*(\sigma) \subseteq Q\}$, where $q_0\Delta^*(\sigma)$ is the extended transition of Δ transition in $\mathcal{A}$ starts from the initial state q_0 of automaton $\mathcal{A}$.

For given alphabets Σ, Σ', and sequence $\sigma \in \Sigma^*$, we denote by $\sigma \downarrow \Sigma'$ the projection σ on Σ' which consists in removing the letters in $\Sigma \setminus \Sigma'$. The projection over Σ' can be naturally extended for the language of an automaton $\mathcal{A}$ by $\mathcal{L}(\mathcal{A}) \downarrow \Sigma' = \{\sigma \downarrow \Sigma' \mid \sigma \in \mathcal{L}(\mathcal{A})\,\}$.

Condition V.1. *When we consider the synchronized product of two automata* $\mathcal{A}_1$, $\mathcal{A}_2$, *we always assume that* $\{Sync(\sigma) \mid \sigma \in \Sigma_1 \cap \Sigma_2\} = AP_1 \cap AP_2$.

$\mathsf{LTL}_f\setminus\mathsf{X}$ is the fragment of *linear temporal logic* without the *next operator* and defines stuttering-invariant properties [29], while the subscript f refers to the fact that we interpret the formulas on finite words [24]. For any $\mathsf{LTL}_f \setminus \mathsf{X}$ formula ϕ over atomic propositions AP, let $\mathcal{A}_\phi$ be an automaton satisfying $\mathcal{L}(\mathcal{A}_\phi) = \{w \in \mathrm{AP}^* \mid w \models \phi\}$. Such an automaton can be constructed by [24]. We make however the following modification: whenever a state of $\mathcal{A}_\phi$ contains a predicate of the form $\mathrm{Sync}(\sigma)$, we label all incoming transitions by σ. Thus, if we denote by AP_ϕ the set of atomic predicates referred to by ϕ, the alphabet of $\mathcal{A}_\phi$ is $\Sigma_\phi = \{\sigma \mid \mathrm{Sync}(\sigma) \in \mathrm{AP}_\phi\}$.

For an automaton $\mathcal{A} := \langle Q, q_0, \Sigma \cup \{\tau\}, \Delta, \mathrm{AP}, L\rangle$, a *finite run* of a given automaton $\mathcal{A}$ is $q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} ...q_{n-1} \xrightarrow{\sigma_n} q_n$ for some $n \in \mathbb{N}$ s.t $\forall i \in [1,..,n]$, $q_{i-1} \xrightarrow{\sigma_i} q_i \in \Delta$. And denote set of possible run of an automaton $\mathcal{A}$ as $\mathrm{Run}(\mathcal{A})$.

For a given finite run $= q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} ....q_{n-1} \xrightarrow{\sigma_n} q_n \in \mathrm{Run}(\mathcal{A})$, define *seq* of run as $\mathrm{seq(run)} := \sigma_1\,\sigma_2...\sigma_n$.

Note that for any given finite run from a given automaton $\mathcal{A}$ i.e run $\in \mathrm{Run}(\mathcal{A})$, the corresponding *seq* belongs in the language of $\mathcal{A}$, i.e $\mathrm{seq(run)} \in \mathcal{L}(\mathcal{A})$.

Definition V.1. *For a given finite run* $= q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} ...q_{n-1} \xrightarrow{\sigma_n} q_n \in$ *Run*$(\mathcal{A})$, *define* trace *of such run as* $trace(run) = L(q_0)L(q_1)L(q_2)...L(q_n) \in AP^*$.

Let us define a trace set for an automaton $\mathcal{A}$ as $\mathrm{Tr}(\mathcal{A}) := \{\mathrm{trace(run)} \mid \mathrm{run} \in \mathrm{Run}(\mathcal{A})\} \subseteq \mathrm{AP}^*$. For $w \in \mathrm{Tr}(\mathcal{A})$, define $\mathrm{run}_w = \{\, \mathrm{run} \mid \mathrm{trace(run)} = w \wedge \mathrm{run} \in \mathrm{Run}(\mathcal{A})\,\}$.

Definition V.2. *Consider the synchronized product of two automata* $\mathcal{A}_1 \parallel \mathcal{A}_2$, *and run* $= (q_{0,1}, q_{0,2}) \xrightarrow{\sigma_1} (q_{1,1}, q_{1,2}), ..., \xrightarrow{\sigma_n} (q_{n,1}, q_{n,2}) \in$ *Run*$(\mathcal{A}_1 \parallel \mathcal{A}_2)$. *The projection of such a run over* Σ_i *for some* $i \in [1,2]$ *is run* $\downarrow \Sigma_i := q_{0,i} \xrightarrow{\sigma'_1} q_{1,i}, ..., \xrightarrow{\sigma'_n} q_{n,i}$ *such that for all* $j \in [1,..,n]$

- *If* $\sigma_j \in \Sigma_i$ *and* $q_{j-1,i} \xrightarrow{\sigma_j} q_{j,i} \in \Delta_i$ *then* $\sigma'_j = \sigma_j$
- *If* $\sigma_j \notin \Sigma_i$ *then* $q_{j-1,i} \xrightarrow{\tau} q_{j,i} \in \Delta_i$ *and* $\sigma'_j = \tau$

Observe that from the synchronized product definition, it is clear that for any given run $\in \mathrm{Run}(\mathcal{A}_1 \parallel \mathcal{A}_2)$, and $A = \{\mathrm{Sync}(\sigma) | \sigma \in \Sigma_1 \cap \Sigma_2\}$, $L_1(q_{j,1}) \cap A = L_2(q_{j,2}) \cap A$ and in fact $|L_1(q_{j,1}) \cap A| = |L_2(q_{j,2}) \cap A| \leq 1$.

The above definition can be extended to runs of products of arbitrary numbers of automata. For an example, if a run as in the form $\mathrm{run} = (q_{0,1}, q_{0,2}, q_{0,3}) \xrightarrow{\sigma_1} ..... \xrightarrow{\sigma_n} (q_{n,1}, q_{n,2}, q_{n,3})$, such that $\forall j \in [0,1,..,n]$, and $\forall i \in [1,2,3]$, $q_{j,i} \in Q_i$ and run $\downarrow (\Sigma_1, \Sigma_2)$ refer to projection of the run to $(q_{0,1}, q_{0,2}) \xrightarrow{\sigma'_1} ..... \xrightarrow{\sigma'_n} (q_{n,1}, q_{n,2})$ and each σ'_j defined as in above definition.

Lemma V.1. *For all automata* $\mathcal{A}_1, \mathcal{A}_2$, *and run* $\in$ *Run*$(\mathcal{A}_1 \parallel \mathcal{A}_2)$, *run* $\downarrow \Sigma_i \in$ *Run*$(\mathcal{A}_i)$ *for* $i \in \{1,2\}$.

Proof. Its trivial from the definition V.2 □

Lemma V.2. *For all automata* $\mathcal{A}$ *and* $w \in$ *Tr*$(\mathcal{A})$, *there exists run* $\in$ *Run*$(\mathcal{A})$ *such that trace*$(run) = w$ *and seq*$(run) \in \mathcal{L}(\mathcal{A})$.

Proof. By definition. □

Definition V.3. *For given atomic predicates sets AP, AP′ and given sequence* $w \in AP^*$, $w \downarrow AP'$ *is the restriction of sequence* w *to AP′.*

Lemma V.3. *For a given* $\mathsf{LTL}_f \setminus \mathsf{X}$ *formula* ϕ *and respective model* $\mathcal{A}_\phi$, *for an atomic predicates set AP,* $AP_\phi \subseteq AP$, *and* $w \in AP^*$, *if* $w \downarrow AP_\phi \vDash \phi$, *then there exists a run* $\in$ *Run*$(\mathcal{A}_\phi)$ *so that trace*$(run) = w \downarrow AP_\phi$.

The above lemma also follows by definition.

Lemma V.4. *Consider an atomic predicates set AP and a* $\mathsf{LTL}_f \setminus \mathsf{X}$ *formula* ϕ *formed over atomic predicates set* $AP_\phi \subseteq AP$. *For all traces* $w \in AP^*$, $w \models \phi$ *if, and only if* $w \downarrow AP_\phi \models \phi$.

Proof. This is immediate from the semantics of LTL since atomic propositions in $\mathrm{AP} \setminus \mathrm{AP}_\phi$ do not influence the satisfaction. □

Definition V.4. *Given any two runs* run_1 *and* run_2 *and atomic predicates set AP, if* $trace(run_1) \downarrow AP = trace(run_2) \downarrow AP$ *then these two runs are equivalent with respect to AP which is denoted as* $run_1 \equiv_{AP} run_2$.

For our theorem, we redefine the semantics of the triples $\langle\phi_1\rangle\mathcal{A}\langle\phi_2\rangle$ as follows.

Definition V.5. *For an automaton $\mathcal{A}$ and $\mathsf{LTL}_f \setminus \mathsf{X}$ formulas ϕ_1 and ϕ_2 with predicate alphabets $AP_{\phi_1}, AP_{\phi_2} \subseteq AP$ and $\mathcal{A}_{\phi_1}$ and $\mathcal{A}_{\phi_2}$, $\Sigma_{\phi_1}, \Sigma_{\phi_2} \subseteq \Sigma$, we define $\langle\phi_1\rangle\mathcal{A}\langle\phi_2\rangle$ as*

$$\forall run \in Run(\mathcal{A}_{\phi_1} \parallel \mathcal{A}),$$
$$trace(run \downarrow \Sigma_{\phi_1}) \vDash \phi_1 \rightarrow trace(run \downarrow \Sigma) \vDash \phi_2$$

Remark V.1. *When considering compositional triples of the form $\langle\phi_1\rangle\mathcal{A}_1\langle\phi_2\rangle$ and $\langle\phi_2\rangle\mathcal{A}_2\langle\phi_3\rangle$, with the intention of showing $\langle\phi_1\rangle\mathcal{A}_1 \parallel \mathcal{A}_2\langle\phi_3\rangle$, we will require that $\Sigma_{\phi_2} = \Sigma_1 \cap \Sigma_2$. Let us recall some simple observations. For all run $\in Run(\mathcal{A}_{\phi_1} \parallel \mathcal{A}_1 \parallel \mathcal{A}_2)$ we have, by lemma V.1, run $\downarrow \Sigma_1 \in Run(\mathcal{A}_1)$, and run $\downarrow \Sigma_{\phi_1} \in Run(\mathcal{A}_{\phi_1})$. If $\langle\phi_1\rangle\mathcal{A}_1\langle\phi_2\rangle$ then $trace(run \downarrow \Sigma_{\phi_1}) \vDash \phi_1$ implies $trace(run \downarrow \Sigma_1) \vDash \phi_2$ and assume for a given run $\in Run(\mathcal{A}_{\phi_1} \parallel \mathcal{A}_1 \parallel \mathcal{A}_2)$, $trace(run \downarrow \Sigma_{\phi_1}) \vDash \phi_1$ then by lemma V.3 there exists a run$' \in Run(\mathcal{A}_{\phi_2})$ such that run $\downarrow \Sigma_1 \equiv_{AP_{\phi_2}}$ run$'$ and by lemma V.2 $seq(run \downarrow \Sigma_1) \downarrow \Sigma_{\phi_2} = seq(run') \in \mathcal{L}(\mathcal{A}_{\phi_2})$.*

Lemma V.5. *Consider automata $\mathcal{A}_1, \mathcal{A}_2$ and formulas ϕ_1, ϕ_2 satisfying $\Sigma_{\phi_2} = \Sigma_1 \cap \Sigma_2$. Assume that $\langle\phi_1\rangle\mathcal{A}_1\langle\phi_2\rangle$ holds. For all run $\in Run(\mathcal{A}_{\phi_1} \parallel \mathcal{A}_1 \parallel \mathcal{A}_2)$, there exists run$' \in Run(\mathcal{A}_{\phi_2})$ and run" $\in Run(\mathcal{A}_{\phi_2} \parallel \mathcal{A}_2)$ which satisfy run" $\downarrow \Sigma_2 =$ run $\downarrow \Sigma_2$ and run" $\downarrow \Sigma_{\phi_2} =$ run$'$.*

Proof. Observe that run$'$ can be obtained as in the previous remark.

Without loss of generality hide all the τ transition in all the runs i.e if a run $=$ run$_3 \xrightarrow{\sigma} (q) \xrightarrow{\tau} (q') \xrightarrow{\sigma'}$ run$_4$ then we work out only for run$_3 \xrightarrow{\sigma} (q') \xrightarrow{\sigma'}$ run$_4$
We construct such a run" by the length of the runs,
we will recall one property which we will use in our construction, for a given run $\in$ Run$(\mathcal{A}_1 \parallel \mathcal{A}_2)$, by remark V.1 $|\text{run}'| \leq |\text{run} \downarrow \Sigma_2|$ because all the common alphabets in $\mathcal{A}_1$ and $\mathcal{A}_2$ fired together in $\mathcal{A}_1 \parallel \mathcal{A}_2$ then those common alphabets $(\Sigma_1 \cap \Sigma_2)$ fired together in the run $\downarrow \Sigma_1$ is captured in $\mathcal{A}_{\phi_2}$ (i.e run$' \equiv_{\text{AP}_{\phi_2}}$ run $\downarrow \Sigma_1$). So that seq(run$'$) $=$ seq(run $\downarrow \Sigma_2) \downarrow \Sigma_{\phi_2}$.

- **Simple case** $|\text{run}_1 = \text{run} \downarrow \Sigma_2| = 1$ and $|\text{run}'| = 1$, run$_1 = q_{0,2} \xrightarrow{\sigma_1} q_{1,2} \in$ Run$(\mathcal{A}_2)$ and run$' = q_{0,\phi_2} \xrightarrow{\sigma_1'} q_{1,\phi_2} \in$ Run$(\mathcal{A}_{\phi_2})$
  - case $\sigma_1 = \sigma_1'$ if this is the case then $\sigma_1 \in \Sigma_1 \cap \Sigma_2$ then the respective run" will be $(q_{0,\phi_2}, q_{0,2}) \xrightarrow{\sigma_1 = \sigma_1'} (q_{1,\phi_2}, q_{1,2}) \in$ Run$(\mathcal{A}_{\phi_2} \parallel \mathcal{A}_2)$
  - case $\sigma_1 \neq \sigma_1'$, this can't be a case, since it is not satisfying seq(run$'$) $=$ seq(run $\downarrow \Sigma_2) \downarrow \Sigma_{\phi_2}$.
- **General case:** here we will construct a run" for given run$'$ and run $\downarrow \Sigma_2$
  w.l.o.g seq(run$_1$ $=$ run $\downarrow \Sigma_2$) $= \sigma_1\sigma_2...\sigma_m$ and seq(run$') = \sigma_{i_1}\sigma_{i_2}...\sigma_{i_n}$ where $n \leq m$ and we know that seq(run $\downarrow \Sigma_2) \downarrow \Sigma_{\phi_2} =$ seq(run$'$).
  $j_1 \in [1, .., m]$ is minimum index number, such that $\sigma_{j_1} = \sigma_{i_1}$, and corresponding run up to executing the σ_{i_1} alphabet looks like run$_{1_{j_1}} = q_{0,2} \xrightarrow{\sigma_1} ... \xrightarrow{\sigma_{j_1-1}} q_{j_1-1,2} \xrightarrow{\sigma_{j_1}} q_{j_1,2}$ and run$'_1 = q_{0,\phi_2} \xrightarrow{\sigma_{i_1}} q_{1,\phi_2}$ then construct run"$_{j_1} = (q_{0,\phi_2}, q_{0,2}) \xrightarrow{\sigma_1} (q_{0,\phi_2}, q_{1,2}) ... \xrightarrow{\sigma_{j_1-1}} (q_{0,\phi_2}, q_{j_1-1,2}) \xrightarrow{\sigma_{j_1} = \sigma_{i_1}} (q_{1,\phi_2}, q_{j_1,2})$. Now find a minimum $j_2 \in [j_1 + 1, m]$ such that $\sigma_{j_2} = \sigma_{i_2}$ and construct run"$_{j_2}$ which we do repeatedly until for j_n we will get run"$_{j_n}$ and then extent run"$_{j_n}$ by just firing the alphabets from $\sigma_{j_n+1}, ..\sigma_m$ we will get run". If we check this constructed run it does satisfies run" $\downarrow \Sigma_2 =$ run $\downarrow \Sigma_2 =$ run$_1$ and run" $\downarrow \Sigma_{\phi_2} =$ run$'$.

□

The theorem V.1 can be stated with above conditions as follows,

Theorem V.2. *For all automata $\mathcal{A}_1$ and $\mathcal{A}_2$, and $\mathsf{LTL}_f \setminus \mathsf{X}$ formulas ϕ_1, ϕ_2, ϕ_3 using the atomic propositions in $AP_{\phi_1} \subseteq AP_1$, $\{Sync(\sigma) \mid \sigma \in \Sigma_1 \cap \Sigma_2\} \subseteq AP_{\phi_2} \subseteq AP_1$, and $AP_{\phi_3} \subseteq AP_2$, $\Sigma_{\phi_1} \subseteq \Sigma_1$, $\Sigma_{\phi_2} = \Sigma_1 \cap \Sigma_2$, and $\Sigma_{\phi_3} \subseteq \Sigma_2$, the following inference rule holds.*

$$\frac{\langle\phi_1\rangle\mathcal{A}_1\langle\phi_2\rangle \qquad \langle\phi_2\rangle\mathcal{A}_2\langle\phi_3\rangle}{\langle\phi_1\rangle\mathcal{A}_1 \parallel \mathcal{A}_2\langle\phi_3\rangle}$$

Proof. Consider any $w \in \text{Tr}(\mathcal{A}_{\phi_1} \parallel \mathcal{A}_1 \parallel \mathcal{A}_2)$, and run $\in$ run$_w$, and assume that trace(run $\downarrow \Sigma_{\phi_1}) \vDash \phi_1$. We need to prove that trace(run $\downarrow \Sigma_2) \vDash \phi_3$.

By lemma V.1 $\forall$run $\in$ run$_w$, (run $\downarrow (\Sigma_{\phi_1}, \Sigma_1)$) $\in$ Run$(\mathcal{A}_{\phi_1} \parallel \mathcal{A}_1)$ and by premise ($\langle\phi_1\rangle\mathcal{A}_1\langle\phi_2\rangle$) trace(run $\downarrow \Sigma_{\phi_1}) \vDash \phi_1$ implies trace(run $\downarrow \Sigma_1) \vDash \phi_2$.

By lemma V.2 and lemma V.3 there exists a run$' \in$ Run$(\mathcal{A}_{\phi_2})$ s.t trace(run $\downarrow \Sigma_1) \downarrow \text{AP}_{\phi_2} =$ trace(run$'$) and by lemma V.1 run $\downarrow \Sigma_2 \subseteq$ Run$(\mathcal{A}_2)$.

It is easy to observe that there is a run" $\in$ Run$(\mathcal{A}_{\phi_2} \parallel \mathcal{A}_2)$ by lemma V.5 such that run" $\downarrow \Sigma_2 =$ run $\downarrow \Sigma_2$ and run" $\downarrow \Sigma_{\phi_2} =$ run$'$.

By premise ($\langle\phi_2\rangle\mathcal{A}_2\langle\phi_3\rangle$) trace(run" $\downarrow \Sigma_{\phi_2}) \vDash \phi_2$ implies trace(run" $\downarrow \Sigma_2) \vDash \phi_3$. I.e trace(run $\downarrow \Sigma_2) \downarrow \text{AP}_{\phi_3} \vDash \phi_3$ then by lemma V.4 trace(run $\downarrow \Sigma_2) \vDash \phi_3$. □

We argue that compositional reasoning is particularly well adapted for verifying SDN protocols. In fact, SDN platforms are distributed systems made of well-separated components, which are the management plane, the controller plane and the data plane. In order to meet our global specifications of isolation and connectivity, each plane has to satisfy a certain property. One of the sources of complexity in verification comes from the fact that several planes are considered in the model. Compositional reasoning allows us to verify properties of each plane separately, and combine these rules to infer the specification for the global system. We explain the application of Theorem V.1 in the next section.
To use Theorem V.1 we need to formally state the specifications for each components in LTL. Finding these properties requires domain knowledge. In fact, one needs to understand with which intentions the system was designed in order to write these formulas.

The management plane and control plane should work together to satisfy the following requirement:

$$\mathsf{I}_1 = \bigwedge_{\substack{1\le i\neq j\le k, x,x'\in \text{Dev}\\ \mathsf{VS}(x)\cap\mathsf{VS}(x')\neq\emptyset\\ v\in \text{Sw}_i, v'\in \text{Sw}_j\\ p,p'\in\text{Pts}}} \begin{array}{l}\Box\big(\text{pos}(x)=(v,p)\wedge \text{pos}(x')=(v',p')\\ \quad\wedge(\text{rpos}_i(x,\text{root}_i)\neq\bot)\rightarrow\\ \quad\Diamond\text{Sync}(c_i,\text{root}_i,(x,\text{root}_j))\big).\end{array}$$

The above formula says that whenever devices x and x' are respectively connected at switch-port pairs (v,p) and (v',p') in respectively domains i and j, if the root switch of domain i has received a MAC learning packet originated from device x, then eventually the root switch will receive a MAC update rule from the controller to forward MAC packets of device x towards root_j. This I_1 captures the fact that the control plane sends appropriate MAC rule updates to the root switch of its domain, thus to the data plane.

$$\mathsf{I}_2 = \bigwedge_{\substack{1\le i\le k, x,x'\in \text{Dev}\\ \mathsf{VS}(x)\cap\mathsf{VS}(x')\neq\emptyset\\ v,v'\in \text{Sw}, p,p'\in\text{Pts}\\ u\in\text{Sw}_i, q,q'\in\text{Pts}\cup\text{Sw}}} \begin{array}{l}\Box\big(\text{pos}(x)=(v,p)\wedge \text{pos}(x')=(v',p')\\ \quad\wedge(\text{rpos}_i(u,x)=q)\\ \quad\wedge(\text{rpos}_i(u,x')=q')\rightarrow\\ \quad\quad\Diamond\text{Sync}(c_i,u,(x,x',q'))\big).\end{array}$$

The above formula states that, given devices x, x' connected at (v,p) and (v',p') respectively, if another switch u has received and forwarded MAC learning packets to the controller witnessed by $(\text{rpos}_i(u,x)=q)\wedge(\text{rpos}_i(u,x')=q')$, then a ping update rule will eventually be sent to switch u to forward packets $\mathsf{ping}(x,x')$ to q'. Thus, I_2 expresses that the control plane sends appropriate ping rule updates to all the switches and thus to the data plane of its domain.

The above requirements should be satisfied jointly by the management and control planes, that is, we would like to establish that $\langle\text{true}\rangle\mathcal{A}^{\text{man}} \parallel \mathcal{A}^{\text{Cont}}\langle \mathsf{I}_1\wedge\mathsf{I}_2\rangle$ should be true. However, this check can still be costly for large topologies. Therefore, we apply again Theorem V.1 to check this compositionally.

We need to introduce another intermediate formula I_3 between the management plane and controller plane. This captures that whenever a management node man_i receives information about a device's position from its controller c_i, eventually, all management nodes man_j forward this information to their respective controller c_j. Thus, the information about a device of domain i is eventually shared with the controller of other domains j.

$$\mathsf{I}_3 = \bigwedge_{\substack{1\le i\le k, x\in\text{Dev}\\ j\in[1,..,k]}} \Box\big((\text{dinfo}_i(x)\neq\bot)\rightarrow \Diamond\text{Sync}(\text{man}_j,c_j,\,(x,\text{dinfo}_i(x),\mathsf{VS}(x))))$$

So we can check the satisfaction of $\langle\text{true}\rangle\mathcal{A}^{\text{man}} \parallel \mathcal{A}^{\text{Cont}}\langle \mathsf{I}_1\wedge\mathsf{I}_2\rangle$ by verifying $\langle\text{true}\rangle\mathcal{A}^{\text{man}}\langle\mathsf{I}_3\rangle$, and $\langle\mathsf{I}_3\rangle\mathcal{A}^{\text{Cont}}\langle\mathsf{I}_1\wedge\mathsf{I}_2\rangle$. Complete description of experimental details are provided in [30].

VI. No-Mobility Scenario Validation

Let us call the Isolation, Connect, and Nomobility as mentioned in section 4.4 as ϕ_I, ϕ_C and ψ respectively. Recall the intermediate formula I_1, I_2, and I_3 We are going to verify, the specified SDN platform in section 3 and 4 satisfies the Isolation and Connect properties under the assumption of Nomobility property.

We are going to compare the usage of Compositional Reasoning (CR) in verifying the SDN platform over the monolithic approach. Monolithic approach consists in verifying the entire system i.e checking whether $\langle\text{true}\rangle\mathcal{A}^{\text{man}} \parallel \mathcal{A}^{\text{Cont}} \parallel \mathcal{A}^{\text{Sw}} \parallel \mathcal{A}^{\text{Dev}}\langle\psi\rightarrow\phi_I\wedge\phi_C\rangle$ is true or false. Compositional rule method 1 (CR method 1) consists in splitting the control plane from the data plane and thus in verifying separately the two automata: $\langle\text{true}\rangle\mathcal{A}^{\text{man}} \parallel \mathcal{A}^{\text{Cont}}\langle\mathsf{I}_1\wedge\mathsf{I}_2\rangle$, and $\langle\mathsf{I}_1\wedge\mathsf{I}_2\rangle\mathcal{A}^{\text{Sw}} \parallel \mathcal{A}^{\text{Dev}}\langle\psi\rightarrow\phi_I\wedge\phi_C\rangle$. Compositional rule method 2 (CR method 2) consists in further splitting the management plane from the control plane and checking three triples: $\langle\text{true}\rangle\mathcal{A}^{\text{man}}\langle\mathsf{I}_3\rangle$, $\langle\mathsf{I}_3\rangle\mathcal{A}^{\text{Cont}}\langle\mathsf{I}_1\wedge\mathsf{I}_2\rangle$, and $\langle\mathsf{I}_1\wedge\mathsf{I}_2\rangle\mathcal{A}^{\text{Sw}} \parallel \mathcal{A}^{\text{Dev}}\langle\psi\rightarrow\phi_I\wedge\phi_C\rangle$.

The results for various sizes of the SDN platform topology and the three methods are shown in Table I. For a given size of SDN platform, we checked all possible SDN constrained topologies as mentioned in Section 3 and 4 and for all possible random choices carried by the devices to connect to their *Access-port* positions. This is the main reason for large memory usage and CPU time spent as indicated in Table I. Within the Nomobility condition, once the device has selected its position and sent its first MAC learning packet, it waits for a timeout. After the first timeout, it sends a second MAC learning packet and waits for the second timeout. After the second timeout, it sends ping packets to all other devices. In this scenario we check that devices sharing a common VS can receive ping packets from each other (i.e. Connect property), and that devices not belonging to this particular VS cannot receive the ping packets from the group and vice versa (i.e. Isolation property). In order to test these Isolation and Connect properties we create the following set of VSes, $\mathsf{VS}=\{\{d_1,d_2\},\{d_3\}\}$, where $\{d_1,d_2\}$ is one VS containing devices d_1 and d_2 and $\{d_3\}$ the other VS containing a single device d_3. We verify that d_3 cannot receive ping packets from devices in $\{d_1,d_2\}$ and vice-versa in order to assert that the SDN platform satisfies the Isolation property. We also verify that eventually d_1 and d_2 can receive ping packets of one another to assert that the SDN platform satisfies the Connect property. For the particular case of no-mobility, the SDN platform satisfies both Isolation and Connect properties. To shortly capture the SDN platform architecture we adopt for the rest of the document the following notations. We denote by $n,(X,Y,Z)$ the set of topologies with *n* domains ($1\le n\le 3$ in our case), where (X,Y,Z) refers to the number of OVSes in each domain. For the single domain with 3, 4, and 5 switches there are respectively two, four and five possible data plane topologies since we restrict to trees. For the case of two domains ($n=2$), the possibilities are $(2,1,-)$, $(3,1,-)$, $(2,2,-)$, and $(3,2,-)$; in this case, there are, respectively, one, two, one, and two possible data plane topologies. With three domains ($n=3$), the possibilities are $(1,1,2)$ and $(1,1,3)$, in which case there are one and two possible data

plane topologies respectively. In our experiments, we enumerate all topologies that match a given template (X, Y, Z) to check exhaustively all possible scenarios. Therefore, the cases with larger numbers of topologies take more time to check.

We ran our experiments on an Intel i7 processor with 13GB of available RAM. Table I compares the monolithic and two compositional approaches, called Method 1 and Method 2, with respect to CPU time and memory usage on SDN topologies of varying sizes.

While the monolithic approach fails to scale above trivial topologies, compositional methods helped us to complete the verification process up to 3 domains in our experiments. However, we still fail to complete the verification process for one domain with 6 switches with both CR methods. The bottleneck during this experiment was checking $\langle \mathsf{I}_1 \wedge \mathsf{I}_2 \rangle \mathcal{A}^{\text{Sw}} \parallel \mathcal{A}^{\text{Dev}} \langle \phi_I \wedge \phi_C \rangle$.

CR method 1 and method 2 provide the same performance up to 2 domains. When the number of domain is 3, CR method 2 gives advantage over the RAM usage and CPU time. Within CR method 1, for 3 domains, much of its CPU and RAM resources are consumed in checking $\langle \text{true} \rangle \mathcal{A}^{\text{man}} \parallel \mathcal{A}^{\text{Cont}} \langle \mathsf{I}_1 \wedge \mathsf{I}_2 \rangle$. Thanks to the additional decomposition between the management plane and the control plane, CR method 2 provides us with better performance. Here the majority of CPU and RAM resources are consumed in checking $\langle \mathsf{I}_1 \wedge \mathsf{I}_2 \rangle \mathcal{A}^{\text{Sw}} \parallel \mathcal{A}^{\text{Dev}} \langle \psi \rightarrow \phi_I \wedge \phi_C \rangle$. The sizes of state space produced in the monolithic method reaches the order of 10^6 for two switches in a single domain, while the compositional approaches produce around 10^4 states for the same architecture. However, the compositional approaches still can not manage the model with 6 switches in a single domain and go out of memory.

n domains,	Monolithic Method		CR Method1		CR Method2	
(X, Y, Z) OVSes	CPU time	RAM used	CPU time	RAM used	CPU time	RAM used
1, (1, −, −)	0.38s	133*MB*	0.85s	134*MB*	1.11s	135*MB*
1, (2, −, −)	166s	723*MB*	1.05s	135*MB*	1.32s	135*MB*
1, (3, −, −)	-	-	6.8s	168*MB*	7s	168*MB*
1, (4, −, −)	-	-	2m 34s	532*MB*	2m 32.4s	531*MB*
1, (5, −, −)	-	-	39m 33s	5002*MB*	39m 40s	5002*MB*
2, (1, 1, −)	-	-	4.07s	145*MB*	1.66s	135*MB*
2, (2, 1, −)	-	-	9.5s	164*MB*	4.05s	164*MB*
2, (3, 1, −)	-	-	1m 29s	505*MB*	1m 10.69s	505*MB*
2, (2, 2, −)	-	-	47.73s	502*MB*	35.77s	502*MB*
2, (3, 2, −)	-	-	15m 21s	4692*MB*	14m 31.3s	4692*MB*
3, (1, 1, 1)	-	-	18m 25s	4292*MB*	4.22s	162*MB*
3, (1, 1, 2)	-	-	36m 11s	6982*MB*	34.62s	472*MB*
3, (1, 1, 3)	-	-	1h 14m 40s	12254*MB*	13m 50s	4482*MB*

TABLE I
MONOLITHIC VS COMPOSITIONAL APPROACHES (WITH NO-MOBILITY)

VII. ISSUE WITH DEVICE MOBILITY

We now present an interesting part of our formal verification experiments. In the last section, we found that the studied SDN platform does satisfy the required specifications, namely the *isolation* property and the *connectivity* property. What if we allow devices to move freely but still use the data plane service to exchange packets? Given this freedom, devices can select their new positions wherever and whenever they want.

This freedom makes the SDN system vulnerable w.r.t. the data isolation property. We identified an error, that is, an execution which violates the isolation property. We can produce the error simply with our monolithic model checking for one domain and two switches, that is, $\langle \text{true} \rangle \mathcal{A}^{\text{man}} \parallel \mathcal{A}^{\text{Cont}} \parallel \mathcal{A}^{\text{Sw}} \parallel \mathcal{A}^{\text{Dev}} \langle \phi_I \rangle$ is violated, where ϕ_I the Isolation property.

We find this error by considering $\mathsf{VS} = \{\{d_1, d_2\}, \{d_3\}\}$ and the following mobility scenario: once first MAC learning packets are exchanged and ping rules are updated, devices d_2 and d_3 swap their respective *Access-port*. After the mobility events, Spin raises an Isolation violation reporting that device d_3 receives a ping packet send by d_1 to d_2.

A. Network Slice Isolation Violation

SPIN reveals a counterexample trace but does not give the probability of its occurrence. In this section, we discuss an illustrative scenario to derive an estimation of the probability.

Let's consider δ as the minimum time needed by the controller to receive a given data plane packet and to respond with a new set of rules to the originating switch (due to the controller queues cdque and mque). Further, let's call ϵ as the time needed by the switch to inform the controller about the possible reconfiguration of the data plane network topology (due to switch queues cque and dque). Finally, let's consider λ as the average number of data packets exchanged per time unit by the switch and devices in the data plane.

With the SDN platform made of a single domain $k = 1$, we have the controller c in the control plane, and three devices $\{d_1, d_2, d_3\}$ and one switch sw which has three ports $\{1, 2, 3\}$ in the data plane. Initially devices d_1, d_2, d_3 are connected respectively to the ports $1, 2, 3$ of switch sw. The management plane applies the following VS of devices $\{\{d_1, d_2\}, \{d_3\}\}$ where $\{d_1, d_2\}$ forms one VS and $\{d_3\}$ is lonely device in another VS. The controller has to ensure that the set of rules it forwards to the data plane (i.e to the switch sw) prevent devices d_3 from receiving packets exchanged between d_1 and d_2. After receiving MAC learning packets from the switch sw, the controller c forwards ping rules to the latter, enforcing that packets from the port 1 should be delivered to port 2 and those from port 2 should be delivered to port 1 and any data packet from the port 3 should be dropped.

When the data plane topology undergoes a transition, such as d_2 and d_3 are exchanging their respective position, it takes the controller c a time interval of $(\delta + \epsilon)$ before being aware of the new situation and actually enforcing new rules. During this lap of time, there are possible scenarios of device d_3 being able to receive packets exchanged between devices d_1 and d_2. For instance, device d_3 occupies now the port 2 of the switch sw and devices d_1, d_2 occupy respectively ports $1, 3$. Due to the communication delay between the switch and the controller, the chance (probability) of device d_3 receiving ping packets delivered by d_1 to d_2 is in the order of $\mathcal{O}(\lambda \times (\delta + \epsilon))$. We are able to reproduce such an Isolation violation with the network simulator called as GNS3.

Thus, the probability of the Isolation violation occurrence is roughly proportional to the packet data rate (i.e. λ) and the overall latency (i.e. $(\delta + \epsilon)$) between the domain controller and the switch, called as the *control link latency*. The first proportionality means that higher bandwidth data paths require faster control plane or smaller *control link latency*. However,

the latter has its own limitation and is very dependent on the SDN architecture. On the studied SDN platform, the controller is centralized so that there is a one-to-one mapping between the controller and the manager in a given domain. This allows for simplifying the synchronization of user's intents and device identity between the two components. Unfortunately, this centralization could induce important *control link latency*. First, it's about communication delay. As uplink communications from different switches within the domain are aggregated when they arrive at the controller, the bandwidth required becomes higher and higher as the number of controlled switches and the number of connected devices increase. Congestion at the controller incoming link could occur if the link was not correctly dimensioned - e.g. a massive arrival of new devices into the data network leads to a massive arrival of MAC-Learning-related PacketIn messages at the controller. Second, it's about processing delay. If the controller was limited in terms of processing power, then its incoming queues could fill up leading to excessive delay and even to loss of control messages (i.e. infinite delay). Furthermore, this also illustrate the fact that the order in which the reading of of messages is important for safety properties for SDN platform.

B. Local Controller Solution

In order to work around previous *control link latency* issues, one possible solution consists in implementing a hierarchical controller architecture with a new controller embedded together with each switch that we call as *local controller* in addition to the existing centralized controller that we call as the *central controller*. Figure 2 illustrates such a controller hierarchical architecture. The *local controller* role simply consists in blocking any outgoing traffic towards a port (e.g. port 2) when the existing device (e.g. d_2) is disconnecting from this port (e.g. a port down status message sent to controllers). It maintains such a blocking rule until a new device (e.g. d_3) is connected to the port and MAC learning messages from this device are processed by the *central controller* and new forwarding rules related to the new device are received by the switch. In order to ensure new rules are actually received by the switch, the *local controller* is implemented as an OpenFlow proxy between the switch and the *central controller*.

As the *local controller* is colocated with the switch, we can assume a zero *control link latency* between the two.

Moreover, the number of devices the *local controller* should deal with is smaller than the one the *central controller* is dealing with by the order of magnitude the number of switches. The variability of control message rate is to be smaller from the *local controller* perspective, which leads to an easier engineering in terms of processing resources. Such a local controller can be modeled by an automata as $\mathcal{A}^{\mathsf{loc.Cont}}$.

C. Local Controller Solution Validation

In this section, we verify that the proposed local controller solution satisfies the Isolation property, i.e $\langle \text{true} \rangle \mathcal{A}^{\text{man}} \parallel \mathcal{A}^{\text{Cont}} \parallel \mathcal{A}^{\text{Sw}} \parallel \mathcal{A}^{\mathsf{loc.Cont}} \parallel \mathcal{A}^{\text{Dev}} \langle \phi_I \rangle$, where ϕ_I is the Isolation property defined in Section V-A and $\mathcal{A}^{\mathsf{loc.Cont}}$ proposed local controller method specified previously. By using the Theorem V.1, we check $\langle \text{true} \rangle \mathcal{A}^{\text{man}} \parallel \mathcal{A}^{\text{Cont}} \langle \mathsf{l}_1 \wedge \mathsf{l}_2 \rangle$ and $\langle \mathsf{l}_1 \wedge \mathsf{l}_2 \rangle \mathcal{A}^{\text{Sw}} \parallel \mathcal{A}^{\mathsf{loc.Cont}} \parallel \mathcal{A}^{\text{Dev}} \langle \phi_I \rangle$ to prove $\langle \text{true} \rangle \mathcal{A}^{\text{man}} \parallel \mathcal{A}^{\text{Cont}} \parallel \mathcal{A}^{\text{Sw}} \parallel \mathcal{A}^{\mathsf{loc.Cont}} \parallel \mathcal{A}^{\text{Dev}} \langle \phi_I \rangle$, where l_1 and l_2 are the intermediate formula defined in Section V-A. The proposed local controller does satisfy the required isolation property. We carry out the experiment for different topologies of the SDN platform. For each topology, we monitor the computational time and the memory usage. Table II provides a synthetic view of these experimental data.

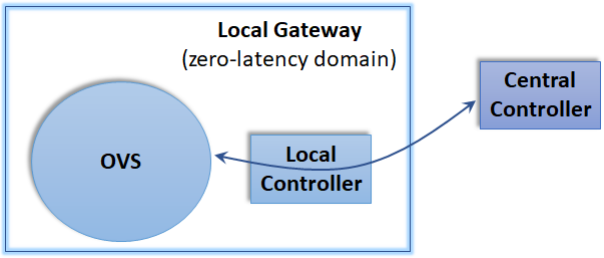

Fig. 2. Local Controller Architecture

n domains, (X, Y, Z) Switches	Compositional reasoning rule $\langle \mathsf{l}_1 \wedge \mathsf{l}_2 \rangle \mathcal{A}^{\text{Sw}} \parallel \mathcal{A}^{\mathsf{loc.Cont}} \parallel \mathcal{A}^{\text{Dev}} \langle \phi_I \rangle$ $\langle \text{true} \rangle \mathcal{A}^{\text{man}} \parallel \mathcal{A}^{\text{Cont}} \langle \mathsf{l}_1 \rangle$ and $\langle \text{true} \rangle \mathcal{A}^{\text{man}} \parallel \mathcal{A}^{\text{Cont}} \langle \mathsf{l}_2 \rangle$	
	Total Computation time	Maximum RAM memory used
1, $(1, -, -)$	4.8sec	188 *MB*
1, $(2, -, -)$	8.4sec	224 *MB*
1, $(3, -, -)$	1min 2sec	448 *MB*
1, $(4, -, -)$	6min 22sec	982 *MB*
1, $(5, -, -)$	1hr 5min 8sec	7245 *MB*
2, $(1, 1, -)$	11sec	214 *MB*
2, $(2, 1, -)$	33sec	395 *MB*
2, $(3, 1, -)$	3min 10.3sec	865 *MB*
2, $(2, 2, -)$	96.5sec	848 *MB*
2, $(3, 2, -)$	25min 13sec	6327 *MB*
3, $(1, 1, 1)$	19min 9.8sec	4291 *MB*
3, $(1, 1, 2)$	36min 44.39sec	6981 *MB*
3, $(1, 1, 3)$	1hr 24min 3.45sec	12254 *MB*

TABLE II
LOCAL CONTROLLER PROPOSAL VALIDATION USING CR METHOD (MOBILITY SCENARIO)

VIII. CONCLUSION

In this document we have demonstrated that it is possible to fully model an SDN architecture including the entire 3 planes, namely the data plane, the control plane and the management plane. It is also possible to include device connectivity and mobility. Although state space can rapidly explode, it is still possible to validate a complex SDN architecture based on a minimum representative model of it.

Within this framework, we have shown that compositional model checking is well-adapted for verifying SDN architectures. It allows one to verify the whole system while verifying each individual component of it. The decomposition of an SDN architecture can be performed, as an illustrative experiment, vertically between planes. This provides more scalability especially when complexity becomes high.

We have demonstrated that it is possible to verify that users' intents are correctly enforced on the data plane in the form of

network slices using model checking. There is a need however to develop an automatic framework to translate high-level intents to concrete network policies in a generic way.

Finally, we have discussed on the trade off between a centralized architecture and a more distributed one and have set up an architecture proposition with regards to network slice isolation issue related to control message transmission latency. Indeed, most of SDN architecture advocates for a centralized controller as this optimizes the interaction with applications and simplifies the related algorithms. This may lead however to some important issues such as the network slice isolation violation revealed in this document. The probability of such an issue is likely to be exacerbated when operating at high data rate. A hierarchical architecture such as the one we propose can solve the issue.

In order to complete this work, we plan to introduce formal synthesis methods aiming at automatically synthesizing the local SDN controller algorithm with regards to some set of safety constraints as tackles in [31]. Other directions are to abstract the system so that traditional supervisory control synthesis of reactive systems can be applied [32].

REFERENCES

[1] M. Boussard, D. T. Bui, L. Ciavaglia, R. Douville, M. L. Pallec, N. L. Sauze, L. Noirie, S. Papillon, P. Peloso, and F. Santoro, "Software-defined lans for interconnected smart environment," in *2015 27th International Teletraffic Congress*, 2015, pp. 219–227.

[2] G. J. Holzmann, *The SPIN model checker: Primer and reference manual*. Addison-Wesley Reading, 2004, vol. 1003.

[3] R. Majumdar, S. Deep Tetali, and Z. Wang, "Kuai: A model checker for software-defined networks," in *2014 Formal Methods in Computer-Aided Design (FMCAD)*, Oct 2014, pp. 163–170.

[4] B. Bingham, J. Bingham, F. M. d. Paula, J. Erickson, G. Singh, and M. Reitblatt, "Industrial strength distributed explicit state model checking," in *2010 Ninth Int. Workshop on Parallel and Distributed Methods in Verification, and Second Int. Workshop on High Performance Computational Systems Biology*, USA, 2010, p. 28–36.

[5] A. Farzan, Y.-F. Chen, E. M. Clarke, Y.-K. Tsay, and B.-Y. Wang, "Extending automated compositional verification to the full class of omega-regular languages," in *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 2–17.

[6] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *9th USENIX Symposium on Networked Systems Design and Implementation*, San Jose, CA, 2012, pp. 113–126.

[7] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *Proceedings of the ACM SIGCOMM 2011 Conference*, 2011, p. 290–301.

[8] T. Nelson, A. Guha, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "A balance of power: Expressive, analyzable controller programming," in *PSACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 2013, p. 79–84.

[9] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A nice way to test openflow applications," in *9th USENIX Conference on Networked Systems Design and Implementation*, 2012, p. 10.

[10] D. Sethi, S. Narayana, and S. Malik, "Abstractions for model checking sdn controllers," in *Formal Methods in Computer-Aided Design*, 2013, pp. 145–148.

[11] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky, "Vericon: towards verifying controller programs in software-defined networks," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 282–293.

[12] K. A., X. Zou, K. Zhou, M. Caesar, and P. Brighten Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, 2013, pp. 15–27.

[13] A. Wang, S. Moarref, B. T. Loo, U. Topcu, and A. Scedrov, "Automated synthesis of reactive controllers for software-defined networks," in *2013 21st IEEE International Conference on Network Protocols, ICNP 2013, Göttingen, Germany, Oct. 7-10, 2013*, 2013, pp. 1–6.

[14] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *ACM SIGCOMM 2012 Conference, SIGCOMM '12, Helsinki, Finland - August 13 - 17, 2012*, 2012, pp. 323–334.

[15] J. McClurg, H. Hojjat, P. Černý, and N. Foster, "Efficient synthesis of network updates," *SIGPLAN Not.*, vol. 50, no. 6, p. 196–207, Jun. 2015.

[16] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev, "Network-wide configuration synthesis," in *Computer Aided Verification*, R. Majumdar and V. Kunčak, Eds., 2017, pp. 261–281.

[17] G. Plotkin, N. Bjørner, N. Lopes, A. Rybalchenko, and G. Varghese, "Scaling network verification using symmetry and surgery," in *43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16, 2016, p. 69–83.

[18] L. Foundation. (2016) Open vswitch. [Online]. Available: https://www.openvswitch.org/

[19] O. N. Foundation. (Dec. 19, 2014) Openflow switch specification - version 1.5.0. [Online]. Available: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.0.pdf

[20] P. Peloso, D. T. Bui, and M. Boussard, "Enforcing users' constraints in dynamic, software-defined networks of devices," in *2017 19th Asia-Pacific Network Operations and Management Symposium*, 2017, pp. 106–111.

[21] L. Foundation. (2016) Open vswitch advanced features. [Online]. Available: http://docs.openvswitch.org/en/latest/tutorials/ovs-advanced/

[22] D. T. Bui, R. Douville, and M. Boussard, "Supporting multicast and broadcast traffic for groups of connected devices," in *2016 IEEE NetSoft Conference and Workshops*, 2016, pp. 48–52.

[23] A. Pnueli, "The temporal logic of programs," in *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, ser. SFCS '77, 1977, p. 46–57.

[24] G. De Giacomo and M. Y. Vardi, "Linear temporal logic and linear dynamic logic on finite traces," in *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, ser. IJCAI '13. AAAI Press, 2013, p. 854–860.

[25] C. B. Jones, "Specification and design of (parallel) programs." in *IFIP congress*, vol. 83, 1983, pp. 321–332.

[26] A. Pnueli, "In transition from global to modular temporal reasoning about programs," in *Logics and Models of Concurrent Systems*, K. R. Apt, Ed., 1985, pp. 123–144.

[27] E. W. Stark, "A proof technique for rely/guarantee properties," in *Foundations of Software Technology and Theoretical Computer Science, Fifth Conference, New Delhi, India, December 16-18, 1985*, pp. 369–391.

[28] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, *Handbook of model checking*. Springer, 2018, vol. 10.

[29] D. Peled and T. Wilke, "Stutter-invariant temporal properties are expressible without the next-time operator," *Information Processing Letters*, vol. 63, no. 5, pp. 243 – 246, 1997. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020019097001336

[30] *A. Majith, O. Sankur, H. Marchand, D.T. Bui, Open access Repository for "Compositional model checking of an SDN platform": Spin source codes*, https://gitlab.inria.fr/anoordhe/drcn2021milan.

[31] G. Kalyon, T. Le Gall, H. Marchand, and T. Massart, "Symbolic supervisory control of distributed systems with communications," *IEEE Transaction on Automatic Control*, vol. 59, no. 2, pp. 396–408, February 2014.

[32] N. Ayeb, E. Rutten, S. Bolle, T. Coupaye, and M. Douet, "Towards an Autonomic and Distributed Device Management for the Internet of Things," in *IEEE 4th International Workshops on Foundations and Applications of Self* Systems*, Umea, Sweden, Jun. 2019, pp. 246–248.